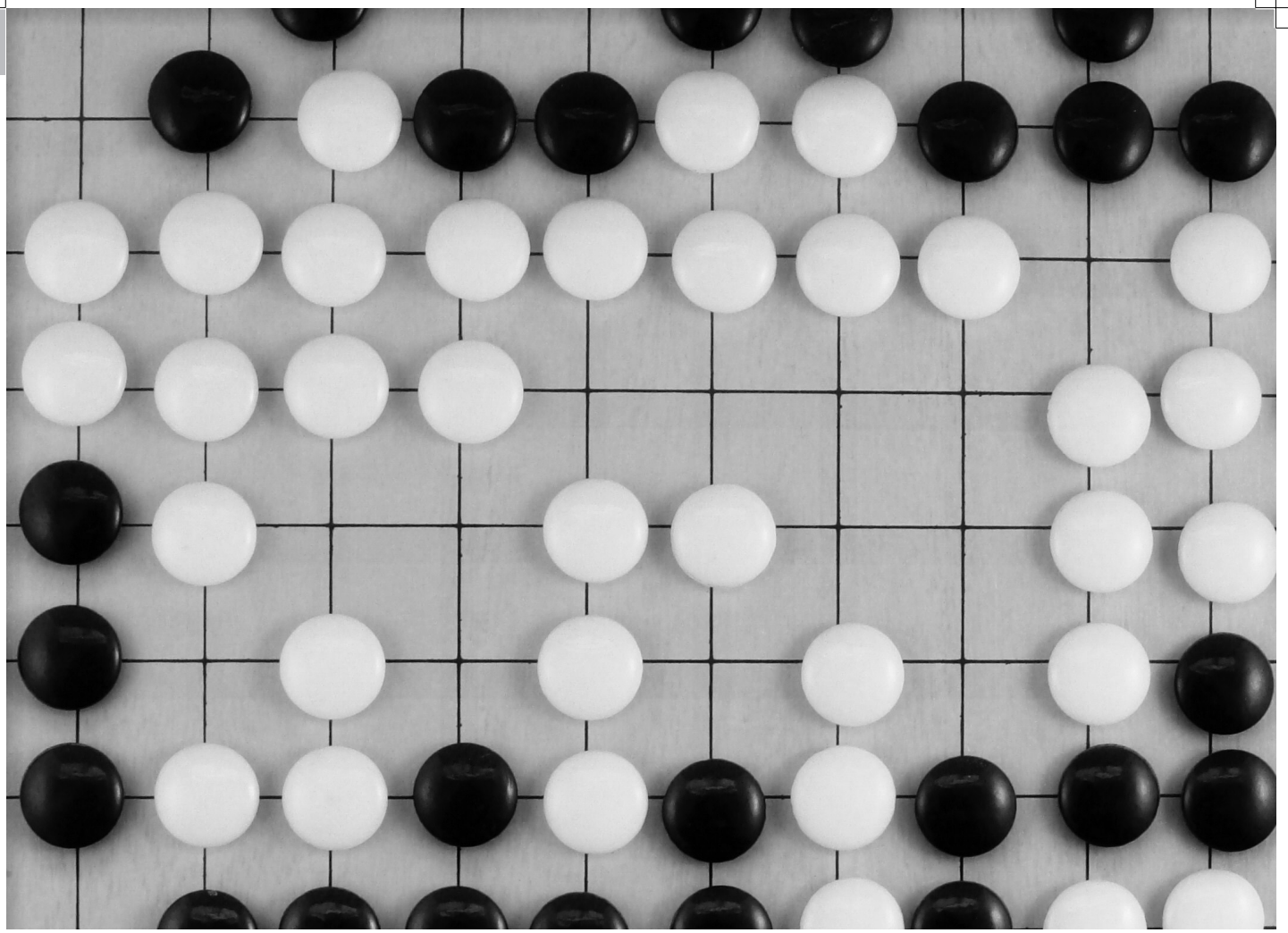

第四章

設計多緒應用程式的八個規則

本書標題很清楚的點出「並行程式設計比起科學，更加接近技藝」，本章提出八個簡單的規則，能夠加入讀者的多緒設計方法工具箱，試著依照使用的先後順序排列規則，但實際上各規則間的順序並非那麼固定，有點像是「不要在游泳池旁奔跑」以及「不要在淺水區跳水」之類的規則，兩個規則都很好，但沒有特定的先後次序。

遵循這些規則，能夠更有機會為應用程式產生最好也最有效率的多緒實作，其中幾點在前幾章中有提過，你可能還有印象，在接下來的章節中，討論設計與開發特定演算法時，會試著提到與這八個規則的關係，盡量不讓這些規則成為只是佔據篇幅的內容。

規則一：找出真正獨立的計算

之前已經提過這個規則，這個規則是一切的關鍵，值得再一次提出來強調，除非執行的運算相互獨立，否則就無法並行執行，很容易舉出現實生活中實現相同目標的獨立活動，例如，DVD 出租店中，訂單被分派給個別員工到庫房找出符合訂單內容的影片，負責尋找經典音樂劇的員工作業並不會影響到其他尋找最新科幻電影的員工，也不會影響尋找熱門影集第二季的員工（假設訂單傳送到庫房前就已經處理好已出借的影片），此外，個別訂單的打包與郵寄也不會影響尋找影片或是處理理其他訂單的程序。

有些情況下會存在無法並行化的特定序列運算，可能是迴圈迭代間存在相依性，或是必須依照特定順序執行步驟，在《那些無法平行》一節中列出了常見的情況。

規則二：在最高層級實作並行

多緒化序列程式碼時有兩種不同的方向：「由下而上」以及「由上而下」。一開始分析程式碼時，會尋找耗費多執行時間的運算熱點，將這些部分平行執行最有機會達到最高效能。

使用由下而上的方式時，直接考慮將熱點多緒化，發生困難時順著應用程式的呼叫堆疊向上尋找包含熱點且能夠平行化的位置，如果熱點位於巢狀迴圈結構的最內層，就由內而外依序檢查各層迴圈，是否有那個層級能夠並行化；即使可以直接並行化熱點的程式碼，也應該看看是否可以在呼叫堆疊較高的位置並行化，能夠提高執行緒運算的粒度。

假設要將影片編碼應用程式多緒化，而熱點發生在計算個別像素，可以試著平行化處理計算單一 frame 中各像素的迴圈，再往上走，可能發現處理各 frame 影像的迴圈可以透過將 frame 分組成獨立的運算並行執行，如果影像編碼程式可以同時處理多部影片，可以將不同串流分派給個別執行緒處理將會是最高層級的並行化。

另一種多緒化的方向是由上而下，先考慮整個應用程式以及各運算的目的（應用程式的各個部分就是為了實現運算目的而結合），如果沒有明顯能夠並行化的部分，將包含熱點的部分運算持續細分成更小的單元，直到找出獨立的運算。

以影像編碼應用程式而言，如果熱點是計算個別像素，由上而下的方式會先考慮處理個別影像串流編碼的部分（包含了像素的計算），如果能夠在這個層級平行化應用程式，就找到了最高的層級，如果不能平行化，朝著計算個別像素的方向向「下」細分為處理個別串流中的單一 frame 以及處理 frame 中的各像素。

這個規則的目的是找出能夠實作並行化且包含熱點的最高層級，一般認為在演算法中的層級愈高也就等同於愈多（獨立）工作，就像是百匯在杯子的愈上層就包含愈多食材一樣，將並行放置在熱點週邊最高的層級是最可能將所有粗質的工作分解分派給個別執行緒的方式。

規則三：儘早規劃擴充性以利用核心數成長

本書寫作時，四核心處理器正成為業界的標準，未來處理器包含的核心只會持續增加，因此，應該在軟體納入對這些處理器的規劃，**可擴縮性**（scalability）是評估應用程式處理系統資源（如核心數目、記憶體大小以及匯流排速度等）以及資料集大小變動的能力（一般是指增加），對於即將到來更多的核心，必須撰寫能夠利用不同核心數量，有彈性的程式碼。

引用 C. Northcort Parkinson 的說法「資料會成長到耗盡可用處理能力」，這表示隨著運算能力提昇（更多核心），需要處理的資料量也隨著增加，總是還有更多的運算需要執行，也許是增加科學模擬的模型真實度、處理 HD 串流而非標準解析度，或是搜尋多個巨大資料庫，只要有了額外的處理能力，總是有人能提供更多需處理的資料。

使用資料分解法設計與實作並行化能提供較有可擴縮性的解決方案，工作分解法會受限於固定的獨立函式或程式片段數量，在每個獨立工作都有對應可使用的執行緒或核心之後，針對更多的核心增加更多執行緒並不會提高應用程式的效能，由於資料量比起應用程式中的獨立運算量更可能成長，資料分解式設計會是最可能提供可擴縮性的方式。

即使應用程式是以執行緒指派獨立函式的方式撰寫，當輸入的工作增加，仍然可以利用更多執行緒，例如有固定收銀台數量的雜貨店，如果老闆買下相鄰的土地，店內空間就成長了一倍，可以預期會有更多工作人員負責相同的工作，額外的油漆工、額外的屋頂工人、額外的地板舖設工人以及額外的水電工，因此，應該要注意因為資料集大小增加而帶來的資料分解可能性，即使是解決方案是以工作分解為基礎，也可以規劃在額外的核心上使用額外的執行緒。

規則四：盡量使用多緒安全函式庫

如果熱點運算能夠透過函式庫呼叫執行，應該強烈考慮使用相當的函式庫函式而非執行手寫程式碼，即使對序列應用程式「重新發明輪子」也從來都不是個好主意，沒有必要重新撰寫已經裝在最佳化函式庫當中的運算。如 Intel Math Kernl Library（MKL）與 Intel Performance Primitives（IPP）之類的許多函式庫，都提供了能夠利用多核心處理器好處的多緒函式。

比起使用多緒函式庫更重要的是，確保所有函式庫的函式呼叫都提供多緒安全，如果將序列程式碼中的熱點用函式庫函式取代，在應用程式中更高的層級仍然有機會分解為獨立運算，當程式中的並行運算包含呼叫函式庫函式時，特別是第三方函式庫，參考與更新函式庫內共享變數的函式可能產生資料競爭，對於所有用於並行執行的函式，都應該檢查函式庫有關多緒安全的說明；如果是自行開發的函式庫函式會被並行執行，要確定函式能夠重進入（reentrant），如果做不到，為了保護對共享資源的存取，就必須加上同步機制。

規則五：使用正確的多緒模型

如果多緒函式庫不足以涵蓋應用程式所有的並行，就必須自行控制執行緒，如果隱式多緒模型（implicit threading model，如 OpenMP 或 Intel Threading Building Block）能夠符合需求就不要使用外顯多緒，外顯多緒雖然在多緒實作上提供較精確的控制，但如果只是多緒化運算密集的迴圈或是不需要外顯多緒提供的額外彈性，就沒有必要耗費額外的精神，實作的複雜度愈高，愈容易犯錯，也愈難以維護程式碼。

OpenMP 以資料分解法為主，特別針對處理大資料集的多緒迴圈，即使應該是程式中唯一採用的多緒型式，仍然會存在其他外部需求（例如老闆或管理上要求的工程規範）使得無法採用 OpenMP，可能需要使用適當的（外顯）模型自行實作多緒化，對於這種情況，建議使用 OpenMP 為並行化建立原型系統，評估可能獲得的效能成長、預期的可擴縮性，以及將序列程式以外顯方式多緒化需要投入的資源。

規則六：不要假設執行順序

在序列運算時，很容易預測程式述句的執行順序，另一方面，各執行緒的執行順序是**非確定性**（nondeterministic），由作業系統排程器控制；這表示沒有可靠的方式能夠預測執行時執行緒的執行順序，也無法知道接下來會執行的執行緒，這麼做主要是為了隱藏應用程式內部的執行延遲，特別是在核心數低於執行緒數的系統上更是如此，如果有執行緒因為需要不在快取中的記憶體或是因為處理 I/O 要求而需要等待，排程器會將執行緒移出，執行其他可以執行的執行緒。

資料競爭是非確定性排程的必然結果，如果假設某個執行緒會在其他執行緒存取共享變數前先寫入適當數值，這個假設可能正確，也可能錯誤，甚至有時正確有時錯誤；幸運的話，在特定平台上每次執行應用程式時，執行緒都能夠以相同的順序執行，系統間所有的差異（磁碟上位元的位置或記憶體速度或是 AC 電源的頻率等等）都可能會影響執行緒排程，依賴執行緒以特定順序執行，卻沒有任何強制機制的程式碼只是很樂觀的認為不會遇到資料競爭或鎖死等問題。

從效能的角度，應該盡量讓執行緒沒有任何限制，就像獵犬或賽馬一般；除非有絕對的必要，不要限制執行順序，程式設計師要判斷真正需要的時機，實作某種型式的同步機制，協調執行緒間的執行順序。

在接力賽中，第一個跑者儘可能的加速，但要得到優勝，二、三與其他位跑者需要等到接到接力棒後才能夠開始在自己分派到的賽程中奔跑，接力棒是連續跑者間的同步機制，控制了比賽各階段間的「執行」。

規則七：盡量使用執行緒私有儲存空間或為資料指定存取鎖

同步是個負擔，除了保證應用程式的平行執行能夠產生正確答案之外，對運算本身沒有任何貢獻，是個必要之惡；即使如此，程式設計師應該盡量減少同步的數量，可以透過使用執行緒私有儲存空間或是互斥的記憶體位置（例如以執行緒 ID 為索引的陣列元素）。

暫時性的工作變數很少在執行緒間共享，應該宣告或配置在執行緒私有空間，持有各執行緒各自運算部分結果的變數也應該是執行緒私有，將部分結果結合到共享位置需要一些同步，確保共享的更新頻率盡量降低能夠減少額外的負擔，如果採用外顯多緒，可以使用執行緒私有空間 API 確保私有資料能夠在執行緒每次執行間或是執行緒內的各函式呼叫時存續。

如果無法使用執行緒私有儲存就必須透過同步機制（例如鎖）協調共享資源的存取，要確保為資料加上適當的鎖，最簡單的方式是使讓鎖與資料有一對一的對應，如果有多個總是一同存取的共享變數，使用單一鎖控制所有包含這組變數的互斥存取。稍後的章節中會討論這種作法的優缺點，以及其他可用的同步技巧，特別是針對需要保護大量資料集合的存取（例如有 10000 個元素的陣列）。

不論採用何種鎖定方式，永遠不要在同一資料物件上加上多個鎖，如同 Segal 定律「有一隻手錶的人知道現在幾點，有兩隻手錶的人不知道現在幾點。」一旦物件有兩個不同的存取保護鎖（lockA 與 lockB），部分程式碼使用 lockA，其他部分程式碼則使用 lockB 控制存取，執行這兩部分程式的執行緒會產生資料競爭，各自都認為自己擁有目標資料的互斥存取權。

規則八：大膽修改演算法獲取更多並行的可能性

不論是序列或並行應用程式，比較效能時的最底限是以時鐘測量的執行時間，選擇演算法時程式設計師會參考漸近執行時間，這個指標大多用於比較應用程式間的相對效能，也就是將其他視為常數，使用 $O(n \log n)$ 的演算法（如快速排序法）會比 $O(n^2)$ 的演算法（如選擇排序法）來得快，而且能得到相同結果。

對於並行應用程式，較好的漸近執行時間的應用程式執行速度也比較快，然而，有時候效能最好的序列演算法並不適合平行化，如果難以將熱點轉換為多緒程式（也無法在呼叫堆疊更高的層級找到包含熱點，能夠並行化的位置），應該考慮採用次佳的序列演算法作為轉換基礎，而非程式碼現實採用的演算法。

例如線性代數中兩個正方形矩陣的相乘，Strassen 演算法有最佳的漸近執行時間（$O(n^{2.81})$），比起一般三層迴圈演算法所需的 $O(n^3)$ 好得多，Strassen 的方式將每個矩陣分為四個區塊（或子矩陣），透過七個遞迴呼叫執行 n/2×n/2 子矩陣的相乘，要平行化這些遞迴呼叫，可以建立執行緒分別執行七個獨立的子矩陣相乘，執行緒數量會以指數成長（就像童謠中來自聖艾維斯的太太、布袋、貓與小貓一樣），隨著子矩陣愈來愈小，新建立執行緒被指派的工作粒度也隨之變小，達到特定大小時，再切換回序列演算法。

平行化矩陣乘法比較簡單的方式是使用 asymptotically inferior triple-nested loop 演算法，有幾種分解矩陣資料的方式（依行、依列或依區塊）並將必要的運算指派給執行緒，可以在最外層使用 OpenMP 指示（pragma）或是使用外顯多緒搭配必要的陣列索引規則實作，比較簡單的序列演算法只需要少量的修改，比起多緒化 Strassen 演算法也幾乎不需要改動結構，更好的是搭配規則四可以使用執行矩陣相乘的並行函式庫。

結語

使用執行緒將序列應用程式轉換為並行版本時應該要牢記這八個簡單的規則，筆者個人透過這八個規則輕易的建立出較強固，也較少多緒問題的並行解決方案，能夠用少量的開發時間就達到最好的效能，相信讀者也做得到。